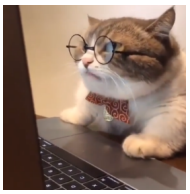
Introducción a la programación - Intento de apunte

2 de julio de 2023



Índice

1. Computadoras, algoritmos, programas y especificaciones

Las definiciones básicas que vamos a manejar a partir de ahora son

- Una **computadora** es una máquina que **procesa información automáticamente**. Esto lo hace siguiendo un programa que está dentro de su memoria interna.
- Un **algoritmo** es una serie de pasos detallados para resolver un problema a partir de unos datos de entrada.
- Un **programa** es la implementación de un algoritmo. Está escrito en un lenguaje de programación y esto es lo que la computadora entiende.

También a lo largo de la materia vamos a trabajar sobre los conceptos de especificación.

Concepto	**Definición**	**Sobre qué actúa**
Especificación/Contrato	Descripción del problema a resolver. Es un contrato entre quien programa una función y quién la va a usar. Define las propiedades de los datos de entrada y de salida	QUÉ problema existe.
Algoritmo	Descripción, dirigida a los humanos, de la solución al problema	CÓMO se soluciona el problema.

1.1. Partes de una especificación/contrato

1. Encabezado
2. Precondiciones - Cláusulas requiere
   - Condiciones sobre los argumentos que recibe el programa que quien programa da por ciertas.
   - Especifica lo que requiere la función para realizar su tarea
   - Evita contradicciones y se asumen que valen todos juntos (conjunción)
3. Postcondiciones - Cláusulas asegura
   - Condiciones sobre el resultado que el programador garantiza (sii el usuario cumplió las precondiciones)
   - Puede haber más de un asegura. Valen todos como conjunción. No pueden contradecirse.
   - Especifica lo que la función asegura que se va a cumplir después de ser utilizada(si las condiciones eran verdades)

Cómo se ve?

```
problema nombre(parámetros) :tipo de dato del resultado{
  requiere etiqueta { condiciones sobre los parámetros de entrada }
  asegura etiqueta { condiciones sobre los parámetros de salida }
}
```

Visualización de cómo debería verse una especificación. Por lo general se escriben en lenguaje formal.

1.2. Parámetros y tipos de datos

- Los **parámetros** son los **datos de entrada** de una especificación. Sus valores se conocen recién al ejecutar el programa.
- Tienen un tipo de dato asociado. Por lo tanto, tienen propiedades que cumplen esos datos de manera innata.

1.3. Beneficios de la especificación - PEC

- **P**revenimos errores.
- **E**ntendemos mejor el problema.
- **C**onstruimos el programa más fácilmente.

2. Lógica Proposicional

Se utiliza en las especificaciones. Sus símbolos, para esta materia, son:

$$True, False, \sim, \wedge, \vee, \Rightarrow, \Leftrightarrow, (,)$$

Las fórmulas atómicas que usaremos para construir las especificaciones son

1. True y False son fórmulas.
2. Cualquier variable proposicional es una fórmula (p,q,r,etc)
3. Si A es una fórmula, $\sim$A es una fórmula.
4. Si $A_1, ..., A_n$ son fórmulas, $(A_1 \wedge A_2..., A_{n-1} \wedge A_n)$ es una fórmula.
5. Si $A_1, ..., A_n$ son fórmulas, $(A_1 \vee A_2..., A_{n-1} \vee A_n)$ es una fórmula.
6. Si A y B son fórmulas, A $\Rightarrow$ B es una fórmula.
7. Si A y B son fórmulas, A $\Leftrightarrow$ B es una fórmula.

2.1. Semántica clásica

En este caso tenemos sólo dos valores de verdad: Verdadero(V) y Falso(F). Cuando sabemos el valor de las variables proposicionales de una fórmula, podemos calcular el valor de verdad de la fórmula(o sea, .[el] total").

2.2. Tablas de verdad clásicas

P	$\sim$ **P**
V	F
F	V

p	**q**	**p** $\wedge$ **q**
V	V	V
F	F	F
V	F	F
F	V	F

p	**q**	**p** $\vee$ **q**
V	V	V
F	F	F
V	F	V
F	V	V

p	**q**	**p** $\Rightarrow$ **q**
V	V	V
F	F	V
V	F	F
F	V	V

p	**q**	**p** $\Leftrightarrow$ **q**
V	V	V
F	F	V
V	F	F
F	V	F

Lo di todo para hacer esas tablas, pero ni idea cómo ponerlas a la derecha.

2.3. Tautologías, contradicciones y contingencias

- Una **tautología** es una fórmula que **siempre** toma el valor **V** para valores definidos de sus variables proposicionales. Ej: $(p \wedge q) \Rightarrow p$ es una tautología

p	**q**	**p ∧ q**	**(p ∧ q) ⇒ p**
V	V	V	V
F	F	F	V
V	F	F	V
F	V	F	V

  Para hacer "la cuenta"hay que mirar la columna 1 y 3.

- Una **contradicción** es una fórmula que siempre toma el valor **F** para valores definidos de sus variables proposicionales. Ejemplo $p\wedge \sim p$

p	**∼p**	$p\wedge \sim p$
V	F	F
F	V	F

- Una **contingencia** es una fórmula que no es ni una tautología, ni un absurdo.

2.4. Teorema: Equivalencia entre fórmulas

Lloro, es el único teorema de la materia :(

1. Doble negación $(\sim\sim p \Leftrightarrow p)$
2. Idempotencia $((p \wedge p) \Leftrightarrow p)$
   $((p \vee p) \Leftrightarrow p)$
3. Asociatividad $((p \wedge q) \wedge r) \Leftrightarrow (p \wedge (q \wedge r))$
   $((p \vee q) \vee r) \Leftrightarrow (p \vee (q \vee r))$
4. Conmutatividad $(p \wedge q) \Leftrightarrow (q \wedge p)$
   $(p \vee q) \Leftrightarrow (q \vee p)$
5. Distributividad $((p \wedge (q \vee r)) \Leftrightarrow ((p \wedge q) \vee (p \wedge r)))$
   $((p \vee (q \wedge r)) \Leftrightarrow ((p \vee q) \wedge (p \vee r)))$
6. Leyes de Morgan $(\sim (p \wedge q) \Leftrightarrow (\sim p \vee \sim q))$
   $(\sim (p \vee q) \Leftrightarrow (\sim p \wedge \sim q))$

2.5. Relación de fuerza

Cuchá, decimos que **A es más fuerte que B** cuando **A⇒B** es una **tautología**. También se puede decir que B es más débil que A.

Ejemplos: Para dados p y q,

1. ¿$p \wedge q$ es más fuerte que p? Sí. Lo vimos en el ejemplo de tautología. Acá A = $p \wedge q$ y B = p.
2. ¿$p \vee q$ es más fuerte que p? No. Se puede hacer la tabla de verdad para comprobarlo. Acá A = $p \vee q$ y B = p.
3. ¿p es más fuerte que $q \Rightarrow p$? Sí. Pero notemos que si q está indefinido y p es verdadero entonces $q \Rightarrow p$ está indefinido. Lo de indefinido viene de la lógica ternaria que veremos más adelante.
4. ¿p es más fuerte que q? No.
5. ¿p es más fuerte que p? Sí
6. La fórmula más fuerte de todas es **False.**
7. La más debil es **True.**

2.5.1. Intercambio de requisitos

Sabiendo las relaciones de fuerza que existen extre los requiere de distintos programas, podemos decidir si podemos(o no) reemplazarlos entre si. Por ejemplo, sean p1 y p2 dos programas que aseguran C. Sea A el requiere que cumple p1 y B el que cumple p2. Si sabemos que A es más fuerte que B, podemos garantizar que todo programa que cumple p2, cumple p1. Pero no vale la vuelta. ¿Por qué? Es más fácil ver esto con conjuntos. El conjunto de soluciones de p1 está contenido en el conjunto de soluciones de p2 (A⊂B). Esto quiere decir que p2 cumple p1 y más. El conjunto de soluciones para el requiere B es más grande. En cambio, si queremos reemplazar el requiere B por el A, vamos a tener casos en donde el programa se indefine porque A al ser un conjunto más chico, tiene menos soluciones.

2.6. Expresiones bien definidas

Se dice que una expresión está **bien definida si todas las proposiciones tienen un valor de T o F.** Pero hay fórmulas que no están bien definidas para algunos valores ejemplo $\frac{x}{y} = 5$ no está bien definida si y = 0. Por esta razón, se introduce la lógica **trivaluada**. Que nos permite decir que está bien definida la siguiente expresión $y = 0 \vee \frac{x}{y} = 5$

2.7. Lógica trivaluada - Secuencial

Ahora tenemos 3 valores de verdad.

1. Verdadero - V
2. Falso - F
3. Indefinido - $\perp$

Se dice secuencial porque se **lee de izquierda a derecha.** La evaluación**termina** cuando **se puede deducir el valor de verdad** aunque **el resto esté idefinido**.

Ahora, nuestros operadores lógicos cambian $\wedge_L$ y $\vee_L$ y se modifican nuestras tablas de verdad.

p	**q**	**p** $\wedge_L$ **q**
V	V	V
F	F	F
V	F	F
F	V	F
V	$\perp$	$\perp$
F	$\perp$	F
$\perp$	F	$\perp$
$\perp$	V	$\perp$
$\perp$	$\perp$	$\perp$

p	**q**	**p** $\vee_L$ **q**
V	V	V
F	F	F
V	F	V
F	V	V
V	$\perp$	V
F	$\perp$	$\perp$
$\perp$	F	$\perp$
$\perp$	V	$\perp$
$\perp$	$\perp$	$\perp$

p	**q**	**p** $\Rightarrow_L$ **q**
V	V	V
F	F	V
V	F	F
F	V	V
V	$\perp$	$\perp$
F	$\perp$	V
$\perp$	F	$\perp$
$\perp$	V	$\perp$
$\perp$	$\perp$	$\perp$

2.8. Cuantificadores: $\exists, \forall$

Su chiste es que tienen un equivalente con la sintaxis de la lógica proposicional. Ej **lo que dicen no necesariamente es cierto** pero es para mostrar cómo se pueden usar.

$$(\forall n : Z)(1 \leq n \leq 10 \Rightarrow n \equiv 0(2))$$

$$(\exists n : Z)(1 \leq n \leq 10 \wedge n \equiv 0(2))$$

La fórmula general para cada uno respectivamente es:

$$(\forall x : Z)(P(x) \Rightarrow Q(x)$$
$$(\exists x : Z)(P(x) \wedge Q(x)$$

Notemos:

- Un cuantificador universal **generaliza la conjunción**

$$(\forall x : Z)(a \leq x \leq b \rightarrow P(x)) \wedge P(b+1) \leftrightarrow (\forall x : Z)(a \leq x \leq b+1 \rightarrow P(x))$$

- Un cuantificador existencial **generaliza la disyunción**

$$(\exists x : Z)(a \leq x \leq b \wedge P(x)) \vee P(b+1) \leftrightarrow (\exists x : Z)(a \leq x \leq b+1 \wedge P(x))$$

Más aún:

- La negación de un cuantificador universal es un cuantificador existencial, y viceversa.

$$\sim (\forall n : Z)P(n) \leftrightarrow (n : Z) \sim P(n)$$

  No es cierto que todos cumplen P sii existe un elemento que no cumple P.

$$\sim (\exists n : Z)P(n) \leftrightarrow (n : Z) \sim P(n)$$

  No existe un elemento que cumple P sii si todos los elemenos no cumplen P.

3. Introducción a la especificación de problemas

Un programa P se considera correcto para la especificación dada por la precondición y la postcondición exactamente cuando se cumple el contrato. O sea, la especificación. Problemas asociados a la especificación.

3.1. Sobre-especificación

Consiste en dar una **postcondición más restrictiva** de la necesaria o en dar una **precondición laxa.** El problema que tra es que limita los posibles algoritmos que resuelven el problema porque impone más condiciones para la salida o amplía los datos de entrada. Ej de sobreespecificación

$$\begin{aligned} &problema\ \ distinto(x : \mathbb{Z}) : \mathbb{Z}\{ \\ &requiere : \{True\} \\ &asegura : \{res = \ x + 1\} \\ &\} \end{aligned}$$

Al asegurar que res = x + 1, estamos limitando las posibles soluciones. Cuando con x $\neq$1 bastaba.

3.2. Sub-especficación

Consiste en dar una **precondición más restrictiva** de lo que se necesita o dar una **postcondición más débil**. El problema que trae es que deja afuera datos de entrada necesarios o ignora condiciones de salida. O sea, vamos a estar frente a soluciones no deseadas. Ej

$$\begin{aligned} &problema\ \ distinto(x : \mathbb{Z}) : \mathbb{Z}\{ \\ &requiere : \{x > 0\} \\ &asegura : \{res \neq x\} \\ &\} \end{aligned}$$

En vez de hacer una precondición tan restrictiva, podemos poner True

3.3. Tipos de datos

Son valores que traen consigo una serie de propiedades.

3.3.1. Básicos

Tipo	Operaciones
Z	+,-,abs(x),*, mod, potencia, división(la algebraica), (des)igualdades
R	+,-,abs(x),*, potencia, (des)igualdades, logaritmos, fracciones, funciones trigon.
$B = \{true, false\}$	!, &&, ||
Char 'x'- 1 carácter	función ord:ord('A')+1=ord('B'), función char: char(ord(n))=n, desigualdades para comparar sus órdenes ord('a') <ord('b').

3.3.2. Tipos enumerados

Tienen una cantidad finita de elementos y cada uno está denotado por una constante.

$$enum\ Nombre\{constantes\}$$

Nombre(del tipo) tiene que ser nuevo, las constantes están separadas por comas, por convención van en mayúscula, ord(a) da la posicicón del elemento en la definición. Inversa - es la función inversa de ord. Ej

Definimos el tipo `Día` así:

```
enum Día {
  LUN, MAR, MIER, JUE, VIE, SAB, DOM
}
```

Valen:

- ord(LUN) = 0
- Día(2) = MIE
- JUE < VIE

3.4. Tipo upla - tupla

Tienen dos o más elementos. Son de cualquier tipo. Ej:

- $Z \times Z$ pares ordenados de enteros
- $Z \times Char \times Bool$ tripla ordenada de un entero, char y booleano.
- $(a_0, ..., a_m, ..., a_k)_m$ es el valor a_m en caso de que $0 \leq m \leq k$. Si no, no está definido.

3.5. Secuencias

Varios elementos del mismo tipo T, posiblemente repetidos, ubicados en cierto orden. Se nota seq< T >. La vacía es <>. Hay secuencias de secuencias también. Seq<seq<T˘.

- $length(a : seq\langle T\rangle) : \mathbb{Z}$ (notación $|a|$)
- $pertenece(x : T, s : seq\langle T\rangle) : Bool$ (notación $x \in s$)
- indexación: $seq\langle T\rangle[i : \mathbb{Z}] : T$
- igualdad: $seq\langle T\rangle = seq\langle T\rangle$
- $head(a : seq\langle T\rangle) : T$
- $tail(a : seq\langle T\rangle) : seq\langle T\rangle$
- $addFirst(t : T, a : seq\langle T\rangle) : seq\langle T\rangle$
- $concat(a : seq\langle T\rangle, b : seq\langle T\rangle) : seq\langle T\rangle$ (notación a++b)
- $subseq(a : seq\langle T\rangle, d, h : \mathbb{Z}) : \langle T\rangle$
- $setAt(a : seq\langle T\rangle, i : \mathbb{Z}, val : T) : seq\langle T\rangle$

3.6. Predicados

Los usamos para **modularizar** las especificaciones.

3.7. Expresiones condicionales

Es una función que elige entre dos elementos del mismo tipo, según una fórmula lógica llamada guarda.

3.8. Modularización

Partimos los problemas más grandes en más chicos(consejo para la vida). Agarramos un problema grande a resolver y proponemos especificaciones auxiliares que podemos reutilizar. Además, al descomponer los problemas, podemos leerlos con menor dificultad.

4. Paradigmas y buenas prácticas

Son formas de pensar un algoritmo que cumpla un especificación. Básicamente cómo encaramos el problema. Comunmente se separan los paradigmas de programación en dos grandes grupos.

Programación Declarativa	**Programación Imperativa**
Se le indica a la máquina el resultado que se desea pero no el cómo	Se declara de manera explícita el proceso a realizar

4.1. Programación declarativa

Declara un conjunto de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema y detallan su solución. Encontramos

- Paradigma Lógico: Los programas se construyen con expresiones lógicas solamente. Ej: PROLOG.
- Paradigma Funcional: Se basa en la composición funcional(matemática). El resultado de un cálculo es la entrada de otro. Ej: LISP, GOFER, HASKELL.

4.2. Programación Imperativa

Describe la programación como una secuencia de instrucciones o comandos que cambian el estado de un programa. Encontramos

- Paradigma Estructurado: Los programas se dividen en bloques que pueden comunicarse o no entre sí. Existen estructuras de control que dirigen el flujo de ejecución. Ej: PASCAL, C, COBOL
- Paradigma Orientado a Objetos: Se basa en la idea de encapsular los estados y comportamientos en objetos. Estos se comunican entre sí por medio de mensajes. Ej: SMALLTALK.

También hay lenguajes multiparadigma que soportan más de un paradigma, valga la redundancia. Ej: PYTHON, JAVA, PHP, .NET

4.3. Buenas prácticas

Básicamente es comentar pero no en exceso y usar nombres declarativos. No repitas código. Y hacer funciones pequeñas que hagan **una sola cosa.**

Los nombres de los archivos de tu proyecto también deben ser claros.

5. Introducción al paradigma funcional

Recordemos: 'Varios algoritmos pueden cumplir la misma especificación'. En el paradigma funcional, pensamos a los **programas como una colección de funciones**. Los datos de entrada se transforman en un resultado.

Ahora los programas son **ecuaciones orientadas** que definen una o más funciones. Además, **ejecutar** corresponde a **evaluar una expresión**. Luego, las ecuaciones orientadas junto con el mecanismo de reducción(en unos párrafos digo qué es) describen algoritmos.

5.1. Ecuaciones

Para determinar el valor final al aplicarse una función, lo que se hace es reemplazar la subexpresión del lado izquierdo, por su definición en el lado derecho. Esto es

---

doble x = x + x
doble (1 + 1) = (1 + 1) + (1 + 1) =>2 + (1 + 1) =>2 + 2 = 4

5.2. Transparencia Referencial

Es la propiedad de un lenguaje que **garantiza** que **el valor de una expresión depende exclusivamente de sus subexpresiones.** Lo que nos da a:

- Cada expresión del lenguaje representa siempre el mismo valor en cualquier parte del programa. **En otros paradigmas, el significado de una variable depende del contexto.**
- Con ella(la transparencia referencial) podemos verificar correctitud (demostrar que se cumple una especificación).

5.3. Formación de expresiones

Tenemos dos tipos

Expresiones Atómicas/-Normales	**Expresiones compuestas**
No se pueden reducir más y **denotan** un valor	Son combinaciones de exp. atómicas.

5.4. Nuestro caso: Haskell

En haskell cuando fijamos el tipo de dato ya está. Usas ese tipo en tu expresión. Decimos que **Haskell es fuertemente tipado**. Veamos un ejemplo sobre cómo evaluaría Haskell este caso suma (resta 2 (negar 42)) 4

---

resta x y = x - y
suma x y = x + y
negar x = -x

$$suma(\ resta\ 2\ (negar\ 42)4$$

$$(\ resta\ 2\ (negar\ 42) + 4$$

$$(2 - (negar\ 42)) + 4$$

$$(2 - (-42)) + 4$$

$$(2 + 42) + 4$$

$$44 + 4$$

$$48$$

Acá se hicieron dos cosas, se uso el mecanismo de reducción y se uso la evaluación lazy.

5.5. Mecanismo de Reducción

Es la forma en la que los lenguajes funcionales evalúan las expresiones.

1. Se elige una subexpresión.
2. Se puede reducir? reduzco.
3. Repito hasta que no haya nada más que reducir

5.6. Órdenes de evaluación en Haskell

Haskell es **lazy**. Perezoso. Se reduce el redex más externo y más a la izquierda para el cual se sepa qué ecuación del programa se debe aplicar. Nota: Hay otros lenguajes que son ansiosos(ej: C, C++, Pascal, Java) y evalúan primero los parámetros y después la función.

5.7. Indefinición

Son las expresiones para las cuales Haskell no encuentra un resultado. Tenemos las funciones **totales** que nunca se indefinen. Ej suc x = x + 1. Y las funciones **parciales**, hay argumentos para los cuales se indefinen. Ej division x y = div x y. Se rompe para y = 0. Veamos ejemplos de cuándo se puede indefinir o no para los mismos valores.

$$(div\ 1\ 1 == 0)\ \&\&\ (div\ 1\ 0 == 0)$$

No se indefine porque ya la primera condición es falsa y Haskell deja de evaluar. Hace cortocircuito.

$$(div\ 1\ 1 == 1)\ \&\&\ (div\ 1\ 0 == 1)$$

Se indefine porque la primera condición es verdadera y haskell tiene que seguir evaluando para llegar a un valor final. Pero la 2da condición se indefine porque la div por 0 no está definida.

$$(div\ 1\ 0 == 1)\ \&\&\ (div\ 1\ 1 == 1)$$

Se indefine a la primera. Se indefine porque la primera condición es verdadera y haskell tiene que seguir evaluando para llegar a un valor final. Pero la 2da condición se indefine porque la div por 0 no está definida.

5.8. Funciones por casos

Para separar casos usamos guardas. Suongamos que queremos definir la función

$$f(n) = \begin{cases} 1 & si\ n = 0 \\ 0 & si\ no \end{cases}$$

```
funcion :: int -> int
funcion n | n == 0 = 1
          | n /= 0 = 0
```

O también podemos usar la palabra reservada 'otherwise'.

```
funcion :: int -> int
funcion n | n == 0 = 1
          | otherwise = 0
```

5.9. Cuidado con el orden

En Haskell hay que tener cuidado cuando las guardas se solapan, recordemos que es lazy. Si encontró algo que ya le da la solución, listo, deja de evaluar.

```
f1 n | n >= 3 = 5
     | n <= 9 = 7
```

```
f2 n | n <= 9 = 7
     | n >= 3 = 5
```

f1(5) = 5 y f2(5) = 7

5.10. Pattern Matching

También se puede usar pattern matching que es básicamente buscar el patrón que cumple tu entrada.

```
f3 0 = 1
f3 n = 0
```

Se pueden combinar pattern matching y guardas.

5.11. Aplicación de funciones

En el paradigma funcional, las funciones son elementos(valores) que se pueden aplicar. Esa es su función básica. Se anota así f :: T1 ->T2 y se lee "f x es una expresión de tipo T2".

5.12. Polimorfismo

Se llama así a una función a la que se le pueden aplicar distintos tipos de datos sin redefinirla. **El comportamiento de la función no depende paramétricamente del tipo de sus argumentos.** Para definir estas funciones se usan variables de tipo.

5.13. Variables de tipo

- Se denotan con minúsculas.
- Denotan tipos, no variables.
- Cuando se invoca la función, se usa como argumento el tipo del valor.

5.13.1. Clases de tipos

- A los conjuntos de tipo se le pueden aplicar un conjunto de funciones.
- Un tipo puede pertener a distintas clases; Los Float son números Num, con orden Ord, de punto flotante Floating.

Algunas clases:

1. `Integral := ({ Int, Integer, ... }, { mod, div, ... })`
2. `Fractional := ({ Float, Double, ... }, { (/), ... })`
3. `Floating := ({ Float, Double, ... }, { sqrt, sin, cos, tan, ... })`
4. `Num := ({ Int, Integer, Float, Double, ... }, { (+), (*), abs, ... })`
5. `Ord := ({Bool, Int, Integer, Float, Double, ... }, { (<=), compare })`
6. `Eq := ({ Bool, Int, Integer, Float, Double, ... }, { (==), (/=) })`

Las clases de tipos se describen como **restricciones** sobre variables de tipos.

```
triple :: (Num t) => t -> t
triple x = 3*x
```

5.14. Parámetros vs Tuplas

Evita el uso de signos de puntuación (comas y paréntesis) y también sólo cambia el tipo de dato que puede recibir una función. Ej Dato: Poner flechitas se llama **currificación** y es en honor al matemático Haskell B.

```
promedio1 :: (Float, Float) -> Float
promedio1 (x,y) = (x+y)/2

promedio2 :: (Float, Float) -> Float
promedio2 x y = (x+y)/2
```

Curry. Es válido buscar un explicación más extensa.

5.15. Funciones binarias: notación prefija vs. infija

Notación prefija	Notación infija
función antes de los argumentos. Ej: suma x y	función entre los argumentos. Ej: x + y

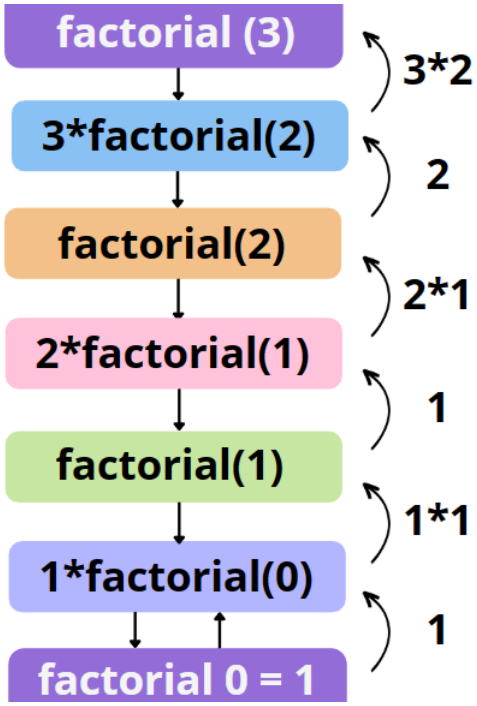
La notación infija se permite para funciones cuyos nombres son operadores. En álgebra vimos que el nombre real de una función definido por un operador • es (•). Ej (+) 2 3.
Una función binaria f puede ser usada de forma infija escribiendo "f".

```
(>=) :: Ord a => a -> a -> Bool
(>=) 5 3 --evalua a True
(==) :: Eq a => a -> a -> Bool
(==) 3 4 --evalua a False
(^) :: (Num a, Int b) => a -> b -> a
(^) 2 5 --evalua 32.0
mod :: (Integral a) => a -> a -> a
5 `mod` 3 --evalua 2
div :: (Integral a) => a -> a -> a
5 `div` 3 --evalua 1
```

6. Recursión

Se habla de recursión cuando en la definición de la función, vuelve a aparecer ella misma y hay un caso base, uno inicial que no usa recursión. Por ejemplo en el caso del factorial en Haskell.

```
factorial :: Int -> Int
factorian n | n == 0 = 1
            | n > 0 = n*factorial(n-1)
```



Intento de esquematización de una función recursiva

Para pensar recursivamente, siempre tenemos que tener un caso base que nos corte la recursión(porque no tiene paso recursivo) y la parte recursiva. Si podemos dar una solución correcta en base a una llamada recursiva entonces, por inducción, todos van a ser correctos.

- Probar por inducción $P(n) : \sum_{i=1}^{n}(2i-1) = n^2$
- Vale para $n = 1 : \sum_{i=1}^{1}(2i-1) = 1^2$
- Supongo que vale $P(n)$, quiero probar $P(n+1)$
- ¿Qué relación hay entre $\sum_{i=1}^{n}(2i-1)$ y $\sum_{i=1}^{n+1}(2i-1)$?

$$\sum_{i=1}^{n+1}(2i-1) = \left(\sum_{i=1}^{n}(2i-1)\right) + 2n + 1$$

- Uso la Hipótesis Inductiva $P(n)$:

$$\sum_{i=1}^{n+1}(2i-1) = n^2 + 2n + 1 = (n+1)^2$$

- ¡¿Pero cómo?! ¡¿Estoy usando lo que quiero probar?!
- Ah, claro... vale $P(1)$ y $P(n) => P(n+1)$, entonces ¡vale para todo n!

- Implementar una función recursiva para $f(n) = \sum_{i=1}^{n}(2i-1)$
- Caso base en Haskell: `f 1 = 1`
- Supongo que ya sé calcular $f(n-1)$, quiero calcular $f(n)$
- ¿Qué relación hay entre $\sum_{i=1}^{n-1}(2i-1)$ y $\sum_{i=1}^{n}(2i-1)$?

$$\sum_{i=1}^{n}(2i-1) = \left(\sum_{i=1}^{n-1}(2i-1)\right) + 2n - 1$$

- Uso la función que sé calcular: $f(n) = f(n-1) + 2n - 1$

  En Haskell: `f n = f (n-1) + 2*n - 1`

- ¡¿Pero cómo?! ¡¿Estoy usando la función que quiero definir?!
- Ah, claro... está definido $f(1)$ y con $f(n-1)$ sé obtener $f(n)$, entonces ¡puedo calcular f para todo n!

Comparación entre inducción y recursión

6.1. Recursión en más de un parámetro

problema *sumatoriaDoble*$(n : \mathbb{Z}, m : \mathbb{Z})$ {
  requiere: $\{(n > 0) \land (m > 0)\}$
  asegura: $\{\text{res} = \sum_{i=1}^{n}\sum_{j=1}^{m} i^j\}$
}

problema *sumatoriaInterna*$(n : \mathbb{Z}, m : \mathbb{Z})$ {
  requiere: $\{(n > 0) \land (m > 0)\}$
  asegura: $\{\text{res} = \sum_{j=1}^{m} n^j\}$
}

Que en Haskell implementado se ve como

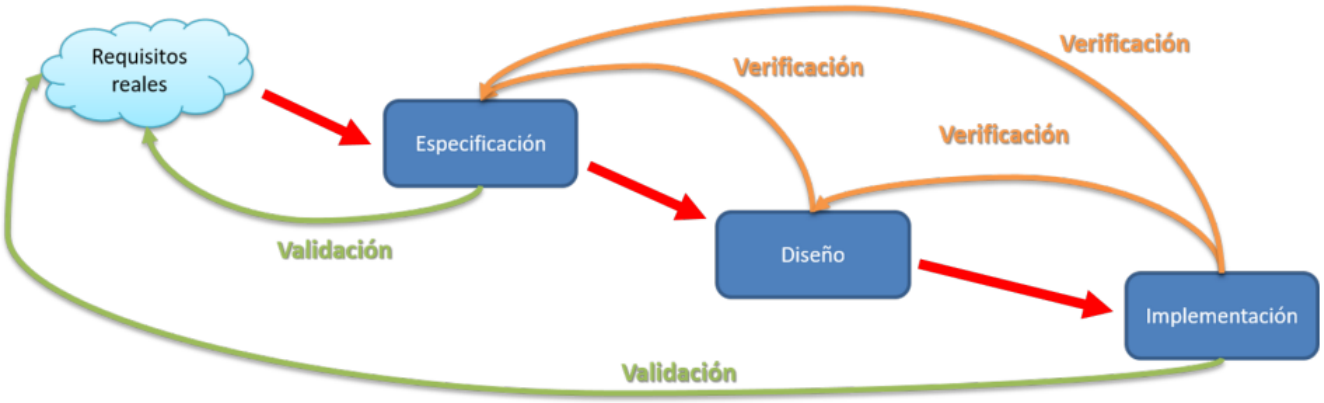
```
sumatoriaInterna :: Integer -> Integer -> Integer
sumatoriaInterna _ 0 = 0
sumatoriaInterna n j = n^j + sumatoriaInterna n (j-1)

sumatoriaDoble :: Integer -> Integer -> Integer
sumatoriaDoble 0 _ = 0
sumatoriaDoble n m = sumatoriaDoble (n-1) m + sumatoriaInterna n m
```

6.1.1. Recursión sobre listas

Listas: Secuencias de elementos **de un mismo tipo**. Los elementos se pueden repetir. Se denota así "[tipo]". En cambio, en un conjunto no tienen orden ni elementos repetidos.Se escribe <set>. Haskell no hace nada para verificar que "type Set a = [a]"se comporte como un conjunto, entonces tenemos que asegurarnos nosotros de eso. Para hacer recursión usamos pattern matching y (x:xs).

7. Introducción a la calidad en software

Vamos a hablar sobre el concepto de **Validación y Verificación(VyV)**.



> V y V es el proceso de comprobar que un sistema de software cumple con sus especificaciones y con su propósito previsto.

La calidad se procura DURANTE todo el proceso de desarrollo. No al final. Esta **se mide** a partir de los siguientes **atributos**(lista no exhaustiva):

- Confiabilidad
- Correción
- Facilidad de Mantenimiento
- Reusabilidad
- Usabilidad
- Interoperabilidad
- Funcionalidad
- Seguridad
- Robustez

Para pensar los conceptos podemos preguntarnos

- Validación: ¿Es el soft. correcto? Hace lo que el usuario quiere?
- Verificación: ¿La manera de hacer el software es la correcta? ¿Hace lo que la especificación declara?

7.1. Diferencias entre falla, defecto y error

Falla	**Defecto**	**Error**
Diferencia entre el resultado esperado y el real	Desperfecto en el sistema que genera más fallas	Equivocación humana. Da lugar a defectos y estos y estos a fallas

7.2. Formas de realizar tareas de VyV para código

Estas tareas se realizan a través del análisis que puede ser:

- Análisis **estático** Es el análisis de una **representación** estática del sistema para descubrir problemas.
- Análisis **dinámico**: Trata con ejecutar y observar el **comportamiento** de un producto.

En la figura podemos ver los distintos tipos de técnicas para cada análisis de VyV. Se menciona al testing y nos da pie para la siguiente sección.

7.3. Testing

Hacer testing es **ejecutar** un producto, hacer que **verifique** los **requerimientos** e **identificar** las diferencias entre el comportamiento real y esperado. Hay distintos niveles de test:

Niveles de test	**Qué hace**
Test de **Sistema**	Abarca todo el sistema. Es el test de aceptación.
Test de **Integración**	Testeamos que los microservicios funcionan bien en conjunto (aparte de individualmente)
Test de **Unidad**	Se testea una porción de código claramente definida

7.4. Anatomía de un Test

Progama bajo Test	**Test Input**	**Test Case**	**Test Suite**
El programa que testeamos.	Es una asignación concreta de valores a los parámetros de entrada para ejecutar el programa bajo test.	Es un programa que ejecuta el programa bajo test usando un test input y chequea si se cumple la condición o no.	Conjunto de casos de Test.

Para elegir datos de prueba, tenemos que buscar aquellos que cumplan la precondición. Y para la condición de aceptación buscamos aquellos que cumplan el asegura.

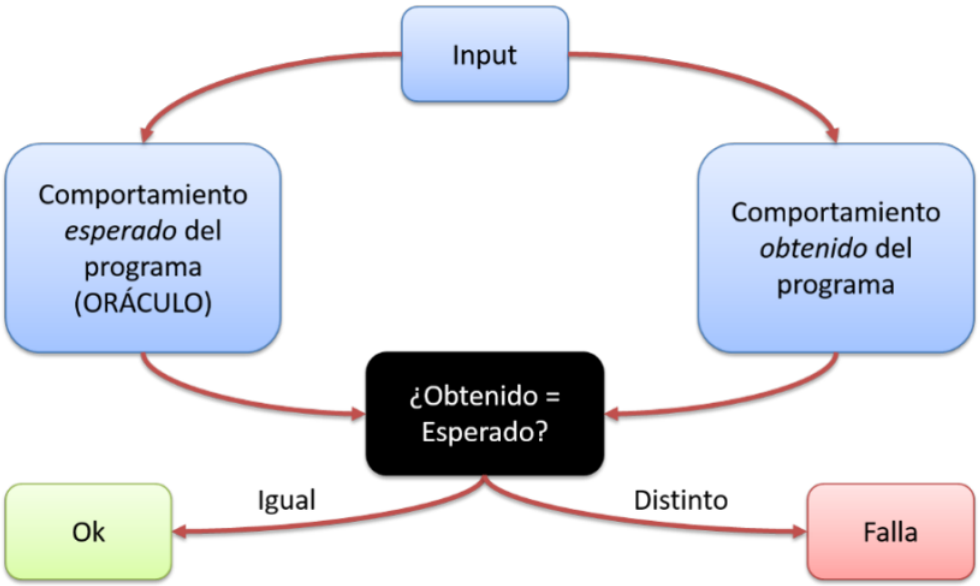
Limitaciones del Testing: “El testing demuestra la presencia de errores, nunca su ausencia” - Edsger Wybe Dijkstra

El amigo era físico, un capo.

Tenemos dos criterios para seleccionar datos de test

Test de Caja Negra	Test de Caja Blanca
Los casos de test se generan analizando la especificación. *No se tiene en cuenta la implementación	Se generan analizando la implementación para determinar los casos de test.

8. Testing de caja negra



Hay problemas asociados al testing, algunos de estos son:

- Sobre el input: ¿Qué datos probar? El objetivo es utilizar **la menor cantidad** de casos de prueba considerando la **mayor cantidad** de escenarios
- Sobre el resultado esperado: ¿Cómo sé qué es lo que espero? Nos fijamos en el contrato/especificación o en el código fuente.
- Sobre el resultado obtenido: ¿Cómo sé cuál fue el resultado obtenido? No suele ser trivial.

Dicho esto, no hay algoritmos que nos propongan casos de test para encontrar todos los errores en cualquier programa. Por eso, recurrimos a los dos criterios para seleccionar datos de test que ya mencionamos antes; Test de Caja Negra y Test de Caja Blanca.
Recordamos: En el test de caja negra obtenemos los datos de test mirando la especificación del programa.

8.1. Método de Partición de Categorías

- Así conseguimos casos de prueba de manera metódica
- Es aplicable a especificaciones formales, semi e informales.

Lo podemos resumir en estos 8 pasos

1. Listar todos los problemas que queremos testear.
2. Elegir uno en particular.
3. Identificar sus **parámetros** o las relaciones entre ellos que condicionan su comportamiento. Los llamaremos genéricamente **factores.**
4. Determinar las características relevantes(categorías) de cada factor.
5. Determinar elecciones(choices) para cada característica de cada factor.
6. Clasificar las elecciones: errores, únicos, restricciones, etc.
7. Armado de casos, combinando las ! = elecciones determinadas para c/ categoría y detallando el resultado esperado en cada caso.
8. Volver al paso 2 y repetir hasta completar todas las unidades funcionales.

8.1.1. Ejemplo del método visto en clase

1. Descomponer el programa en unidades funcionales: Acá enumeramos todas las operaciones y funciones que se van a poner a prueba. Por lo general, se arranca con las funciones que son utilizadas por otras.

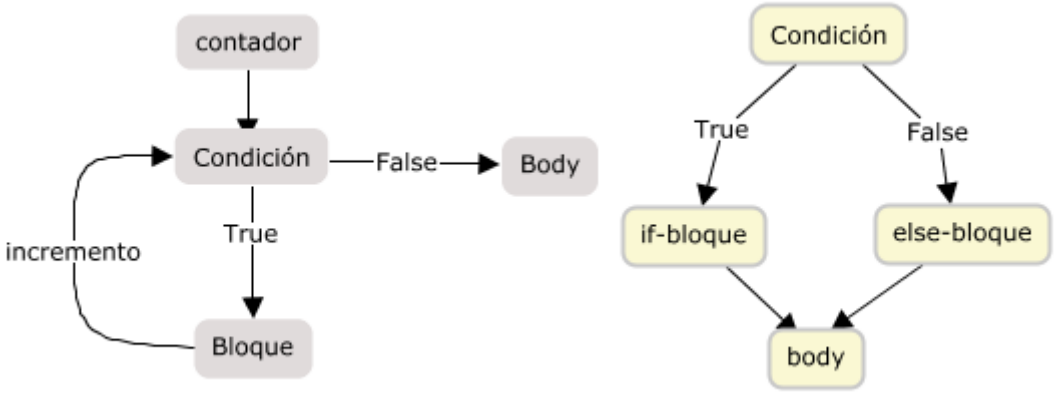
   [imagen del programa espermutacion de la teorica]

   Acá tenemos dos funciones para testear: esPermutacion y cantidadDeApariciones. -El ejemplo seguía en la teórica y no lo copié-

9. Test de Caja Blanca

Se usa para complementar al test de caja negra. De esta manera, nos aseguramos un buen cubrimiento de nuestro programa y podemos saber la confiabilidad de los tests utilizados(Criterio de adecuación del Test de Caja Negra). Los tests de caja blanca **son test estructurales**. Se derivan a partir de **la implementación del programa**.
Para realizar los test, usamos **Control-Flow Graphs**(CFG) que es una representación gráfica del programa. Es **independiente** de las entradas. Cuanta más partes son cubiertas(del programa), más probabilidades de encontrar una falla.

9.1. Control-Flow Patterns



Los CFG siguen básicamente esta estructura. El de la izquierda es un while CFG pero el del for se descompone igual.

9.2. Criterios de adecuación

¿Cómo sabemos que un test es **suficientemente bueno**? Usando los criterios de adecuación, que son unos predicados que toman un valor de verdad para una tupla <programa, test suite>. Usualmente se expresa en forma de una regla del estilo: "Todas las sentencias deben ser ejecutadas."

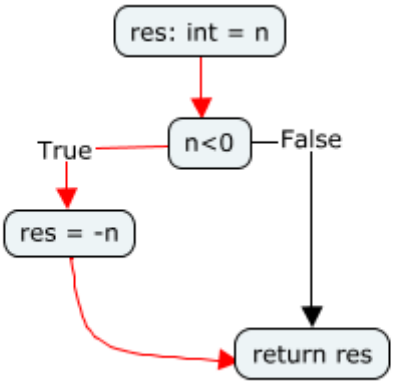
Ej 1: Tenemos la regla 'Cada nodo(sentencia) en el CFG debe ser ejecutado al menos una vez por algún test case.' La idea detrás de esta regla es que un defecto en una sentencia sólo puede ser revelado ejecutando el defecto. La **cobertura de sentencias** del test se calcula como:

$$\frac{\text{cantidad de nodos ejercitados}}{\text{cantidad de nodos}}$$

Ej 2: Tenemos la regla 'Todo arco en el CFG debe ser ejecutado al menos una vez por algún test case'. La idea detrás de esta regla es que si recorremos todos los arcos, recorremos todas las sentencias. Cubrir sentencias $\subseteq$ Ccubrir arcos. La **cobertura de arcos** del test se calcula como:

$$\frac{\text{cantidad de arcos ejercitados}}{\text{cantidad de arcos}}$$

Notemos: cubrir arcoscubrir sentencias (Marcado con rojo). **Puedo cubrir todas las sentencias pero no los arcos.**



Para el test de case de abajo cubrimos todas las sentencias pero no así los arcos(flechas).

Test case 1	
Entrada	-3
Salida esperada	3

Ej 3: Tenemos el criterio de adecuación 'Cada decisión(el conjunto arco-True y arco-False) en el CFG debe ser ejecutado'. En los cubrimientos de decisiones queremos testear por cada if-statement(o cualquier cosa que cambie el flujo del código) el caso True y el caso False. La **cobertura de decisión(branches)** del test se calcula como:

$$\frac{\text{cantidad de decisiones ejercitadas}}{\text{cantidad de decisiones}}$$

Nota: El cubrimiento de decisiones no implica el cubrimiento de los arcos del CFG(preguntar en clase).

Ej 4: Cubrimiento de Condiciones Básicas Una condición básica es una fórmula atómica que compone una decisión. En este caso, el criterio de adecuación es 'cada condición' básica de cada decisión en el CFG debe ser evaluada a verdadero y falso al menos una vez. Cobertura:

$$\frac{\text{cantidad de valores evaluados en cada condición}}{\text{2*cantidad de condiciones básicas}}$$

Veamos una aplicación para tenerlo más claro: Si tenemos if(a&&b). Un test suite que ejercita 'a = b = true' y 'a = false, b = true' cubre ambos branches del if-else. **Pero no alcanza el cubrimiento de decisiones básicas ya que falta b = false.** Generalizando, **Branch(decisiones) coverage NO implica cubrimiento de condiciones básicas.**

Dato de color: Para ser aprobado, todo software que controla un avión necesita ser testeado con cubrimiento de branches y condiciones básicas.

Ej 5: Cubrimiento de caminos. Su criterio de adecuación es 'cada camino en el CFG debe ser transitado por al menos un test case'. Por lo general, no se usa porque es poco práctico. En un bucle while los caminos no están acotados Cubrimiento:

$$\frac{\text{cantidad de caminos transitados}}{\text{cantidad total de caminos}}$$

10. Programación imperativa

Dentro de los paradigmas de programación[1] encontramos a la programación imperativa. Donde ahora los programas **no son funciones necesariamente**. Estos pueden:

- Devolver más de un valor.
- Hay nuevas formas de pasar argumentos.
- los programa son conjuntos de variables, funciones y procedimientos.

[1]Son formas de pensar un algoritmo. Tienen asociados conjuntos de lenguajes y nos hacen encarar la programación según ese paradigma.

- las variables finales deberian resolver el problema.

En comparación a la programación funcional, hay un nuevo concepto de variables.

- Hay posiciones de memoria.
- Cambian explícitamente su valor a lo largo de la ejecución del programa.
- Se pierde la transparencia referencial[2]

Hay una nueva operación llamada **asignación**. Con esta se cambia el valor de una variable. Además, las funciones ya no pertenecen a un tipo de datos y en vez de usar recursión, se usa la **iteración.** Se nos presenta, también, un nuevo tipo de datos; **el arreglo**. Con algunas de sus propiedades:

- Son secuencias de valores de **un sólo tipo**.
- Su **longitud** es **fija**.
- Se puede acceder directamente a una posición

Además, un programa en el paradigma imperativo es un conjunto de tipos, funciones y procedimientos.

10.1. Python

Es el lenguaje que vamos a usar en la materia. Aunque también soporta parte del paradig. de objetos y del funcional. Datos:

- Es un lenguaje interpretado[3]
- Es un lenguaje polimórfico. Tiene tipado dinámico[4] y también acepta fuertemente tipado. [5]
- Se ejecuta de arriba hacia abajo.

10.1.1. Aclaración sobre item 4 y 5

Algorithm 1 Ejemplo de tipado dinamico

```
x = 2
x = "perro"
```

Algorithm 1 no da error porque no se explicitó el tipo de dato. En cambio, en algorithm 2 sí hay un error al ejecutarlo porque explicitamos el tipo int.

Algorithm 2 Ejemplo de fuertemente tipado

```
x:int = 2
x = "perro"
```

10.2. Variables

En este paradigma, las variables son **nombres asociados a un espacio de memoria. Pueden tomar distintos valores** a lo largo de la ejecución de un programa.

[2]La transparencia referencial, referida a una función, indica que se puede determinar el resultado de aplicar esa función solo observando los valores de sus argumentos. Ahora, en la prog. imperativa pueden haber variables globales que modifican tu resultado esperado.

[3]No hay compilación del programa a instrucciones en lenguaje máquina. Se ejecuta línea por línea.

[4]Una variable puede cambiar su tipo de dato.

[5]Una vez declarado el tipo de variable, ya no se puede cambiar.

10.3. Instrucciones

- Asignación
- Condicional
- Ciclos
- Procedimientos: Funciones que modifican sus argumentos.
- Retorno de control

10.4. Asignación (=)

Con ella modificamos el valor de una variable (var = expr;). Es una operación asimétrica. Del lado izquierdo tenemos el nombre y del derecho una expresión del mismo tipo que la variable. Efecto de la asignación

1. Se evalúa la expresión de la derecha y se obtiene un valor.
2. Ese valor se copia en el espacio de memoria de la variable.
3. El resto de la memoria no cambia.

10.4.1. Semántica de la asignación

Sea e es una expresión cuya evaluación no modifica su estado, tenemos entonces

```
#estado a
```

$$v = e;$$

$$v == e@a \land z_1 = z_1@a \land \cdots \land z_k = z_k@a$$

Donde $z_1 \cdots z_k$ son todas las variables del programa en cuestión, distintas de v, que están definidas hasta ese momento

10.5. Retorno de control: Return

Termina la ejecución de una función, retorna el control a su invocador y devuelve el valor de la expresión como resultado.

Las otras variables se supone que no cambian y por convención no se dice nada. Pero si la expresión e es la invocación a una función que recibe parámetros por referencia, puede haber más cambios, aunque al menos sabes que es cierto que:

$$v == e@a$$

Está en la poscondición o asegura de la asignación.

10.6. Alcance de una variable

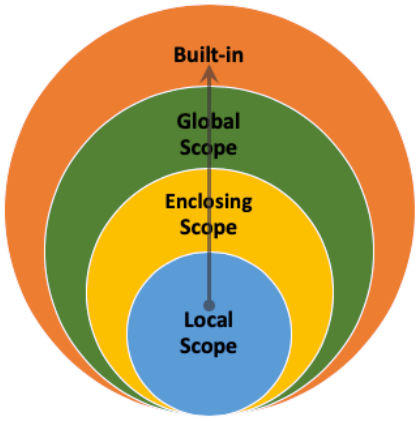
Hay de los tipos

Global	**Local**
Son declaradas fuera del bloque de la función y se puede acceder a ella desde cualquier línea del código	Sólo es reconocida dentro del bloque donde fue declarada.

Una variable sólo es válida dentro del bloque (función/procedimiento) donde fue declarada. Al terminar el bloque, la variable se destruye.

10.6.1. Python tiene 4 niveles de visibilidad

1. Local: Dentro de una función
2. No local / Enclosed: Declarada dentro de una función que tiene otra función
3. Global: Declarada dentro del cuerpo principal del programa
4. Integrado/Built-in: Palabras reservadas de Python (def, print etc)



10.7. Transformación de estados

Se define **estado** de un programa a **los valores de todas sus variables en un punto de su ejecución.** Luego, un programa es una **sucesión de estados.** El resto de las instrucciones modifican el flujo de ejecución.

Algorithm 3 Ejemplo de transformación de estados

```
def ejemplo() -> int:
    x:int = 0  #estado 1 x == 0
    x = x + 3  #estado 2 x == 3
    x = 2 * x #estado 3 x == 6
    return x
```

Podemos pensar que cada instrucción define un nuevo estado, al que a su vez podemos ponerle un nombre para identificarlo y así referirnos a una variable dentro de ese estado en particular. Usamos la siguiente notacion:

`nombre_de_variable@nombre_de_estado`

Algorithm 4 Ejemplo de transformación de estados 2

```
def suc(x:int) -> int:
    #estado a
    x = x + 2
    #estado b; vale x == x@a + 2 "x vale lo que valia en @a + 2"
    x = x - 1
    #estado c; vale x == x@b - 1 "x vale lo que valia en @b - 1"
    return x
```

10.7.1. Argumentos de entradas de funciones

En los lenguajes imperativos, en general, existen dos tipos de pasajes de parámetros.

Pasaje por valor	Pasaje por referencia
Se crea una **copia local** de la variable dentro de la función	**Se maneja directamente la variable.** Los cambios realizados dentro de la función le afectarán también afuera.

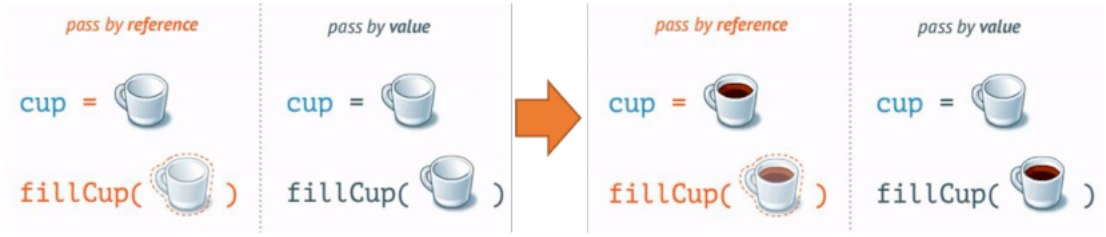


Fig 1: En la figura de la derecha vemos cómo se modifica la variable fuera de la función cuando se pasa su referencia. **La expresión con la que se realiza la invocación debe ser una variable. Debe tener una posición de memoria asociada.**

Conceptualmente, existen tres tipos de pasajes de parámetros.

Entrada(in)	**Salida(out)**	**Mixto (inout)**
Al salir de la función/procedimiento, la variable **mantiene su valor original**	**Dentro de la f/p se le asigna un nuevo valor**. No debería ser leído, su valor original, por la f/p.	Su valor inicial sí importa y se modifica.

Bruno en una práctica dijo que los valores out NO son parámetros que toma una función.

Ejemplo de especificación definiendo el tipo de pasaje. Notemos que no hay un tipo de dato de retorno declarado. Eso es porque ahora, podemos definir **procedimientos**, que recordemos, modifican pero no devuelven un valor. Notemos que en la implementación de python no está el return".

Problema duplicar(**inout** x: seq $< T >$){
Modifica: x
Asegura : { x tendrá todos los de x@pre y además se le concatenará otra copia de x@pre a continuación. }
Asegura : { x tendrá el doble de longitud de x@pre. }
}
Su respectiva implementación en python

```
def duplicar(x:list):
    x=*2
```

Otro ejemplo
Problema duplicar2(**in** x: seq $< T >$) seq $< T >$){
Asegura : { res será una copia de x y además, se le concatenará otra copia de x a continuaciòn }
Asegura : {res tendrá el doble de longitud de x. }
}
Su respectiva implementación en python

```
def duplicar2(x:list)->list:
    y:list= x.copy()
    y*=2
    return y
```

Otro ejemplo
Problema duplicar3(**in** x: seq $< T >$, **out** y: seq $< T >$) {
Asegura : { y será una copia de x y además, se le concatenará otra copia de x a continuación }
Asegura : {y tendrá el doble de longitud de x. }
}
Su respectiva implementación en python

```
def duplicar3(x:list , y:list):
    y.clear() #porque no nos interesa su valor original
    y = x.copy()
    y = y*2
```

10.7.2. Condicionales

Vamos a pensar a los condicionales desde la transformación de estados

- Todo condicional tiene su precondición y su postcondición P_{if} y Q_{if}
- Cada bloque también tiene sus pre y post condiciones.

$$\triangleright Estado\ P_{if}$$

$$if(B):$$

$$\triangleright Estado\ \ P_{uno}$$

$$uno$$

$$\triangleright Estado\ \ Q_{uno}$$

$$else:$$

$$\triangleright Estado\ \ P_{dos}$$

$$dos$$

$$\triangleright Estado\ \ Q_{dos}$$

$$\triangleright Estado\ Q_{if}$$

$$\triangleright (B \wedge estado\ Q_{uno}) \vee (\neg B \wedge estado\ Q_{dos})$$

Después del IF, se cumplió B y Q_{uno} o, no se cumplió B y Q_{dos}

*Mención a los bucles que no voy a agregar

10.7.3. Interrumpiendo ciclos: Break

En la materia no lo recomiendan usar porque saca declaratividad al código y dificulta leer programas. Break lo que hace es romper un bucle.

10.7.4. Ciclos y transformación de estados

En en un programa imperativo, cada instrucción transforma un estado

Con las transformaciones de estado podemos hacer una ejecución simbólica de los programas. Podemos pensar a los ciclos como una instruccion, con un estado previo y otro posterior.

10.8. Arreglos

Son otro tipo de estructura de datos

- secuencia **fija** de valores del **mismo tipo**.[6]
- Tienen asociada una posición de memoria. La variable es el nombre del array. NO los elementos que contiene.
- Si quiero acceder a uno de sus elementos, uso corchetes y se accede de manera directa.[7]

Si en python queremos acceder a su dirección de memoria y cantidad de elementos, podemos usar la función

```
array.buffer_info()
```

10.9. Tipos Abstractos de Datos(TAD)

Es un modelo que usamos para **definir** valores y las operaciones **que se pueden realizar sobre ellos.**

- Se llama abstracto porque no es necesario conocer cómo están implementadas las operaciones o cómo es su representación interna para poder usarlos.

Ej; un TAD que ya vimos son las listas. Porque:

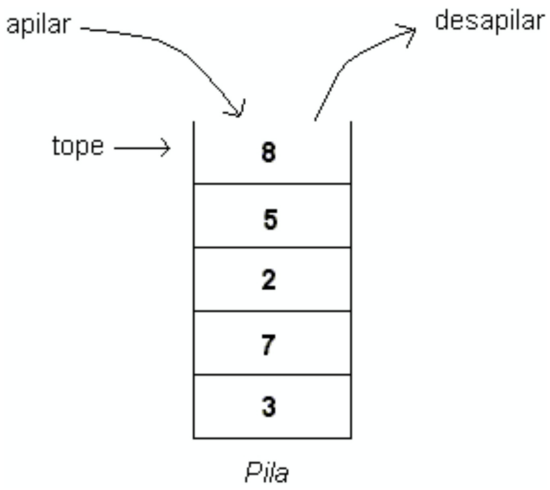
- Se definen como una serie de elementos consecutivos.
- Tiene operaciones asociadas: append, remove, index, etc.
- Desconocemos cómo se guarda la información dentro del tipo.

[6] En python su largo es variable. Pero en general, en los lenguajes de prog además de declarar el tipo se fija su longitud.
[7] Si fuera una lista, primero se acceden a todos los elementos anteriores.

10.10. Pilas

Es una lista de elementos de la que se puede extraer **el último elemento insertado**. Las pilas también son **listas LIFO** - Last in ->First out. Las operaciones básicas que tiene son:

- Apilar: Ingresa un elemento a la pila.
- Desapilar: Saca el último elemento insertado.
- Tope: Devuelve **sin sacar** el último elemento insertado.
- Vacia: Retorna verdadero si está vacía.

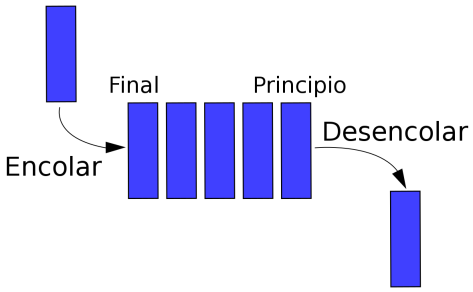


En python podemos pensar a las listas como pilas y usar las operaciones append, pop(desapilar), tope(a[-1]) y len(a)==0 devuelve True si está vacía. Ej de la vida: Una pila de platos acumulados, no podemos sacar el que está debajo de todo sin sacar todos los anteriores primero. Una góndola de supermercado, los elementos más faciles de sacar son los que se colocaron a lo último.

10.11. Cola

Una cola es una lista pero del estilo FIFO- First in ->First out. Sus operaciones básicas son:

- Encolar: ingresa un elemento a la cola.
- sacar: saca el primer elemento insertado.
- varia: retorna verdadero si está vacía.



Ejemplos de la vida real: Una fila en la parada de bondis, una fila en la caja de supermercado una fila en la cabina de peaje.
Si queremos modelar colas en python, tenemos las funciones básicas append(encolar), pop(desencolar), len(a)==0 true si está vacía.

10.12. Diccionarios

Son una estructura de datos donde tenemos pares clave-valor

- Las claves deben ser inmutables - los valores pueden variar
- La clave es un identificador único
- Los diccionarios son mutables, se pueden modificar.
- NO tienen un orden específico. Es decir, no hay garantía de que se respete el orden en el que yo fui declarando las cosas.

  Las operaciones que encontramos son, entre otras; agregar un nuevo par clave valor, eliminar un elemento, modificarlo, verificar que existe una clave guardada, obtener todas las claves y obtener todos los elementos.

10.12.1. Manejo de Archivos

También pueden pensarse mediante la abstracción que nos brindan los TADs. Necesitamos una operación que nos permita abrir un archivo, otra para leer sus lineas y otra para cerrar el archivo. Tenemos

```
archivo=open(path al archivo, modo en el que se abre, encoding)
contenido=archivo.read()
archivo.close()
```

También hay operaciones básicas: read, readline, readlines, write, writelines. Que este intento de apunte te acompañe.